1. Consider the following grammar.
   \[
   \begin{align*}
   T & \rightarrow X \\
   X & \rightarrow SX \mid S \\
   S & \rightarrow \{ \text{ print } P \} \\
   S & \rightarrow \{ \text{ = id } P \} \\
   S & \rightarrow \{ \text{ loop id X } \} \\
   P & \rightarrow F \mid id \mid \text{ num} \\
   F & \rightarrow \{ \text{ fn } P \ P \}
   \end{align*}
   \]
   (a) Construct the LR(0) automata (CFSA) for the grammar.
   (b) Construct the SLR(1) parse table for the grammar.

2. Consider the following grammar.
   \[
   \begin{align*}
   S & \rightarrow A : B \& \\
   A & \rightarrow a \& \mid B \\
   B & \rightarrow aB \mid a
   \end{align*}
   \]
   (a) Construct the LR(1) Automata for the grammar.
   (b) Merge the LR(1) states and convert the LR(1) automata to LALR(1).
   (c) Parse the string a&:aaa& based on the LALR(1) automata. Show the stack, the input, and the actions taken.
   (d) Discuss the property of the grammar in terms of LL(k), SLR(k), LR(k), and LALR(k).

3. Give example grammars for the following specifications.
   (a) A grammar that is not SLR(5) but is SLR(6).
   (b) A grammar that is LR(1), not LALR(1), and is LALR(2).
   (c) A grammar that has reduce-reduce conflict in SLR(1), but is LALR(1) and LR(1).

4. Prove that if a grammar is LL(1), then it is definitely LR(1).

5. Consider the following grammar.
   \[
   \begin{align*}
   S & \rightarrow \text{id} = L \\
   L & \rightarrow L \ N \mid \epsilon \\
   N & \rightarrow \text{num}
   \end{align*}
   \]
   This grammar will generate id = num*. Our goal is to add all the numbers in the input into a list and assign the list to the identifier. Write an attribute grammar to do this. You need to design proper attributes and define the actions on the attributes using pure attribute grammar. You can assume that there is an “addList (list, num)” function which will add the number to the list and return then new list. You can also assume that there is a “createList ()” function which creates an empty list.

6. Consider the following grammar.
   \[
   S \rightarrow TC
   \]
T → TAB | AB
A → Aa | a
B → Bb | b
C → xy

(a) This grammar will generate (a+b)+ xy. Write an attribute grammar to compute the total number of a’s and store it as x.val (val is an attribute for the grammar) and the total number of b’s and store it as y.val. No global variables. You will not be able to write a synthesized attribute grammar, but you can have a left-inherited attribute grammar.

(b) Rewrite the attribute grammar to allow the assignment of x.val and y.val being done at parsing time. To achieve the goal, you have to reference to the stack. Please use Stack[top–i] to reference the i-th element below the top in the stack.

7. Consider the attribute grammar we discussed in class. This attribute grammar is L-inherited and we need to go into the stack to obtain the type information in L.type to assign them to all the id’s. Rewrite the grammar so that it becomes a synthesized attribute grammar and can be evaluated fully from bottom up.

D → T L  L.type = T.type
T → int   T.type = integer
T → real  T.type = real
L → L1, id L1.type = L.type, addtype (id.entry, L.type)
L → id    addtype (id.entry, L.type)